アプリケーション互換性とTCB削減を両立するプロセスベースの信頼実行環境

呉 国駿[1,a)]　飯田 圭祐[1,b)]　竹腰 開[1,c)]　深井 貴明[2,d)]　品川 高廣[1,e)]

概要：信頼実行環境（Trusted Execution Environment; TEE）は，クラウド環境においてホストを信頼することなくアプリケーションの機密性と完全性を保護できる技術である．TEE を広く活用するためには，信頼計算基盤（Trusted Computing Base; TCB）を小さく抑えつつも，既存のアプリケーションがそのまま動作する互換性を確保する必要がある．しかし，従来のエンクレーブ型 TEE はアプリケーション互換性に課題がある一方，機密仮想マシン（Confidential Virtual Machine; CVM）はゲスト OS 全体が TCB に含まれるという問題がある．本研究では，アプリケーション互換性と TCB 削減を両立する新たな TEE の抽象化として，機密プロセス（Confidential Process）を提案する．Confidential Process は，仮想マシンではなくプロセスを基本的な保護対象とする TEE であり，システムコールを通じてホスト OS とやり取りする．ハードウェアによる隔離基盤として CVM を用いることで既存アプリケーションのバイナリを無改変で実行可能にするとともに，TEE 内には信頼されたユーザープロセスと最小限のランタイムのみを配置することで TCB を大幅に削減する．AMD SEV-SNP 上に実装したプロトタイプでは，既存の ELF バイナリを無改変で実行できることを確認し，TCB を約 14,000 行に抑えた．また，性能評価の結果，プロセスの起動から終了までの時間を従来の CVM と比べて約 6.8 分の 1 に短縮できることを確認した．

キーワード：Trusted Execution Environment，Confidential Virtual Machine，Process Abstraction

1. はじめに

信頼実行環境（Trusted Execution Environment; TEE）は，クラウド環境においてホストを信頼することなくアプリケーションの機密性と完全性を保護できる技術として注目を集めている．TEE は CPU を信頼の基点（root of trust）とすることで，OS，ハイパーバイザ，あるいは管理者が侵害された状況においても計算の機密性と完全性を保証する．こうした性質から，TEE はプライバシ保護を伴うデータ分析，機械学習推論，Confidential Container といった安全なクラウドコンピューティングの用途において有望視されている．

しかし，TEE を広く活用するためには，信頼計算基盤（Trusted Computing Base; TCB）を小さく抑えつつも，既存のアプリケーションがそのまま動作する互換性を確保する必要がある．従来の TEE には，このアプリケーション互換性と TCB の削減との間にトレードオフが存在する．Intel SGX に代表されるエンクレーブ型 TEE は，小さな TCB で強固な隔離を実現する一方，エンクレーブ内のコードはシステムコールを直接発行できず，利用可能なライブラリも限られるため，アプリケーション互換性に課題がある．一方，機密仮想マシン（Confidential Virtual Machine; CVM）は，既存のアプリケーションをゲスト OS ごと TEE 内で実行することで，高い互換性を確保できる．しかし，ゲスト OS 全体が TCB に含まれるため，信頼すべきコードが増大し，攻撃対象領域も拡大する．実際，CVM の実行環境やそのインターフェースを悪用する攻撃が複数報告されている [1], [2], [3], [4].

このトレードオフに対し，先行研究は主に二つの方向から取り組んできた．一つ目は，既存のアプリケーションを SGX エンクレーブ内で実行するアプローチである．Haven [5] や Occlum [6] はエンクレーブ内にライブラリ OS を組み込み，SCONE [7] や Panoply [8] はエンクレーブ内に shim layer を置いてシステムコールを仲介する．これらの取り組みは一定の成果を示したが，アーキテクチャ上の

[1] 東京大学
7-3-1 Hongo, Bunkyo, Tokyo 113-8654, Japan
[2] 国立研究開発法人産業技術総合研究所
1-3-1 Kasumigaseki, Chiyoda, Tokyo 100-8921, Japan
[a)] wu@os.is.s.u-tokyo.ac.jp
[b)] iida@os.is.s.u-tokyo.ac.jp
[c)] takekoshi@os.is.s.u-tokyo.ac.jp
[d)] takaaki.fukai@aist.go.jp
[e)] shina@is.s.u-tokyo.ac.jp

ⓒ2026 Information Processing Society of Japan

制約が残るため，アプリケーション互換性には依然として課題がある．二つ目は，CVM の TCB を削減するアプローチである．Gramine-TDX [9] は最小限のライブラリ OS を用い，Veil [10] や Wallet [11] は特権分離によって実効的な TCB を削減している．しかし，これらの手法は実行可能なアプリケーションに制約があるほか，ゲストカーネルを CVM 内に残している．したがって，既存アプリケーションの互換性を維持しながら，TEE 内の汎用ゲスト OS を排除して TCB を削減することは，依然として課題となっている．

本論文では，アプリケーション互換性と TCB 削減を両立する新たな TEE の抽象化として，**機密プロセス（Confidential Process）**を提案する．Confidential Process は，仮想マシンではなくプロセスを基本的な保護対象とする TEE であり，システムコールを通じてホスト OS とやり取りする．ハードウェアによる隔離基盤として CVM を用いることで，エンクレーブ型 TEE のアーキテクチャ上の制約を回避し，既存アプリケーションのバイナリを無改変で実行可能にする．一方，TEE 内には信頼されたユーザープロセスと最小限のランタイムのみを配置し，システムコールの処理を可能な限りホスト OS に委譲することで，TCB を大幅に削減する．Confidential Process は単に CVM を小型化したものではなく，TEE の保護対象を仮想マシンからプロセスへと変更する新たな抽象化である．

この新たな抽象化の実現可能性を評価するため，我々は AMD SEV-SNP 上に Confidential Process を Rust で実装した．プロトタイプは，TEE 内ランタイム，KVM 上に構築した独自の VMM，およびホスト側のプロキシから構成され，83 個のシステムコールに対応している．実装にあたり，システムコールが扱う状態に基づいて，TEE 内で処理すべきもの，ホストに委譲できるもの，および両者で処理を分担するものを分類する基準を明らかにした．また，信頼できない OS による Iago 攻撃 [12] の一種として，Confidential Process に特有のファイルディスクリプタの再利用を突く攻撃を新たに特定し，その対策を実装した．

プロトタイプでは既存の ELF バイナリを無改変で実行できることを確認し，TCB を約 14,000 行に抑えた．LMbench [13] を用いた評価では，TEE 内で完結する操作は Linux と同等以上の性能を示した一方，ホストに委譲するシステムコールの実行時間は Linux の約 85 倍であった．Nginx を用いた評価では，Confidential Process は Linux を動作させた CVM（以下，Linux CVM）の 51%から 65%のスループットを達成した．また，プロセスの起動から終了までに要する時間は 574 ms であり，Linux CVM と比べて約 6.8 分の 1 に短縮できた．

本論文の貢献は以下のとおりである．

- 信頼境界をプロセスレベルで定義する新たな TEE の抽象化として，Confidential Process を提案した．
- システムコールを TEE 内で処理すべきかホストに委譲できるかを判定する基準を示した．
- Confidential Process に特有の Iago 攻撃を特定し，その対策を示した．
- AMD SEV-SNP 上にプロトタイプを実装し，既存アプリケーションの無改変実行，TCB のコード量，実行性能，およびライフサイクルコストを評価した．

本論文の構成は，まず 2 章で Intel SGX と CVM，および AMD SEV-SNP について述べる．3 章で Confidential Process の設計を，4 章でその実装を述べる．5 章で評価結果を示し，6 章で関連研究について述べる．最後に，7 章で本論文をまとめる．

2. 背景

2.1 Intel SGX

Intel SGX [14] は，代表的なエンクレーブ型 TEE である．ユーザープロセスのアドレス空間の一部をエンクレーブと呼ばれる保護領域として隔離する．エンクレーブ内のコードとデータは Enclave Page Cache（EPC）と呼ばれる保護された物理メモリ上に置かれ，CPU によるアクセス制御とメモリ暗号化によって，OS やハイパーバイザを含むエンクレーブ外のソフトウェアからの参照や改ざんが禁止される．

エンクレーブ内のコードは非特権モードでのみ動作し，システムコール命令を含む特権的な操作は実行できない．このため，OS の機能を利用するには，エンクレーブを抜けてエンクレーブ外の信頼できないコードに処理を依頼する必要がある（OCALL）．このようなエンクレーブ遷移は TLB のフラッシュなどを伴うため高コストである．また，エンクレーブ内で同時に実行できるスレッド数は作成時に固定され，OS がエンクレーブのメモリを複製できないため `fork` の実現も難しいなど，アーキテクチャ上の制約が多い．

2.2 機密仮想マシン

機密仮想マシン（CVM）は，仮想マシン（VM）を単位として TEE を提供する．代表的な実装として AMD SEV-SNP [15]，Intel TDX [16]，Arm CCA [17] がある．CVM の脅威モデルでは，ハイパーバイザ，ホスト OS，クラウドの管理者は信頼されず，CVM のメモリとレジスタ状態はそれらから保護される．

メモリの保護は，具体的には暗号化と完全性保護の二つの機構によって実現される．まず，メモリは VM ごとの鍵によってハードウェアで透過的に暗号化され，ハイパーバイザやホスト OS からは平文を参照できない．さらに，物理ページごとの所有者や対応するゲスト物理アドレスをハードウェアが管理し，メモリアクセスのたびに検査する．これにより，ハイパーバイザによるゲストページの書き換

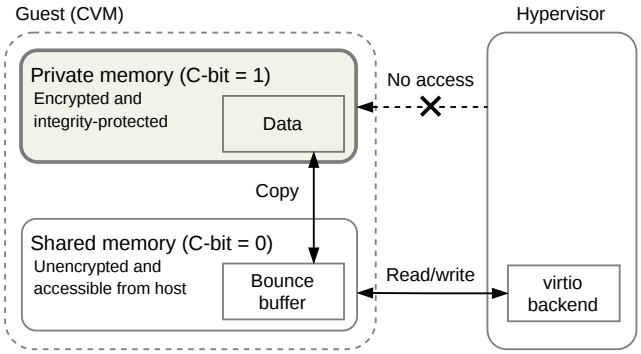

図 1 SEV-SNP におけるプライベートメモリと共有メモリ．ハイパーバイザはプライベートメモリ（緑色）にアクセスできず，CVM とハイパーバイザのデータのやり取りは共有メモリ上のバウンスバッファを介して行われる．

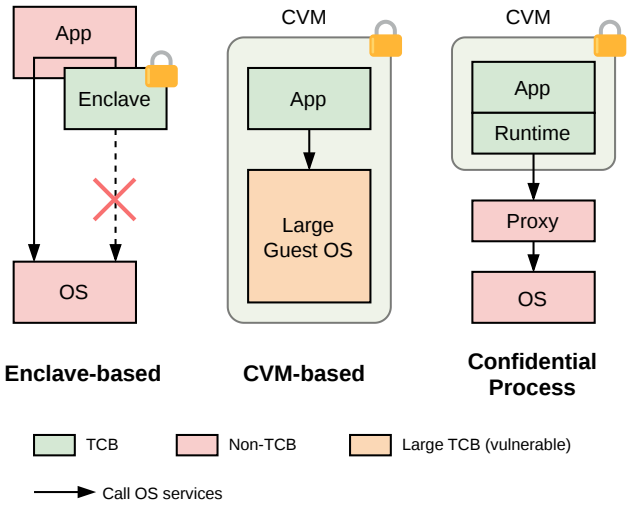

図 2 三つの TEE のアーキテクチャの比較．

えや別アドレスへの再マップといった完全性への攻撃も防がれる．

一方，ゲスト内のファームウェア（OVMF など），カーネル，ユーザー空間のソフトウェアはすべて TEE 内で動作し，TCB に含まれる．通常の VM と同様に CVM 内でゲスト OS は特権モードで動作するため，カーネルを CVM 向けに一部改変すれば，ユーザープログラムは無改変で実行可能である．

2.3 AMD SEV-SNP

本節では，本論文の設計上重要な SEV-SNP の機構として，プライベートメモリと共有メモリの区別，およびゲストとハイパーバイザ間の通信について述べる．

2.3.1 プライベートメモリと共有メモリ

ゲストはページテーブルエントリの C-bit によって，各ページを暗号化されたプライベートメモリとして扱うか，暗号化されない共有メモリとして扱うかを指定する．プライベートメモリについては，ハイパーバイザは平文を読むことも内容を書き換えることもできない．このため，ハイパーバイザやホストとデータをやり取りするには共有メモリを介する必要がある．ページをプライベートと共有の間で切り替えることも可能だが，高コストなハードウェアデータ構造の更新を伴うため，頻繁に行うには向かない．そこで Linux ゲストでは，デバイス I/O のデータをいったん共有メモリ上のバウンスバッファにコピーしてから受け渡す（**図 1**）．このように，外部とのデータのやり取りにはプライベートメモリと共有メモリ間のコピーが本質的に伴い，無視できないオーバーヘッドであることが指摘されている [18]．

2.3.2 レジスタ状態の保護と VMEXIT

SEV-SNP では，VMEXIT 時にゲストのレジスタ状態が暗号化された VM Save Area（VMSA）に保存され，ハイパーバイザからは参照できない．このため，CPUID 命令や MSR へのアクセスなど，従来ハイパーバイザがゲストの状態を参照してエミュレートしていた操作は，ゲスト内で #VC（VMM Communication）例外を発生させる．ゲストの#VC ハンドラは，必要な情報だけを Guest-Hypervisor Communication Block（GHCB）と呼ばれる共有ページに書き込み，VMGEXIT 命令で明示的にハイパーバイザへ制御を移す．すなわち CVM では，ハイパーバイザはゲストの状態を直接参照できず，ゲストが明示的に渡した情報のみに基づいて処理を行う．

3. Confidential Process

Confidential Process の設計は，次の二つの原則に基づく．一つ目は，TEE 内にはプロセスと最小限のランタイムのみを置くということである．ランタイムには最低限必要な機能だけを実装し，ホストの OS カーネルを適宜利用することで TCB を最小化する．二つ目は，標準的なシステムコールに基づくインターフェースでホストとやり取りするということである．システムコールの意味論に基づいて，ホストから返された結果の妥当性を検証する．これらの原則の上で，Confidential Process は無改変のアプリケーション・バイナリを実行できることを目標としている．

図 2 に，三つのアプローチのアーキテクチャの違いを示す．エンクレーブ型 TEE は最小限の TCB を保つ一方，OS のサービスを通常のように呼び出すことができず，無改変のバイナリをそのまま動かすことができない．また，専用のライブラリ OS やコンパイラツールチェーンを使ったとしても，抽象化しきれないアーキテクチャの制約が残るのが実情である．CVM 型の TEE は完全なアプリケーション互換性を実現するが，ゲスト OS 全体を信頼領域に取り込むため TCB が大きくなる．特に汎用 OS は，デバイスドライバや割り込み処理など，ハードウェアを扱うための複雑なコンポーネントが多く，その分攻撃対象領域も広い [18]．Confidential Process では，TEE 内ランタイム

ⓒ2026 Information Processing Society of Japan

は通常の OS カーネルと同様にゲストの特権モードで動作し，プロセスが発行したシステムコール命令を直接捕捉するとともに，ページテーブルや例外を TEE 内で管理する．このため，エンクレーブ型 TEE よりも高い互換性を実現可能である．一方，捕捉したシステムコールの処理自体は可能ならば TEE 外のホストプロキシに委譲するため，ゲスト OS をまるごと TEE 内に置く必要がなく，CVM 型 TEE よりも TCB を大幅に削減できる．さらに，ホストとのインタフェースを主にシステムコールに限定することで，仮想マシンの複雑なインターフェースに起因する攻撃対象領域を削減できる．

3.1 脅威モデル

本システムでは，標準的な CVM の脅威モデルを採用する．すなわち，CPU とその暗号化エンジンは信頼する一方，OS やハイパーバイザを含むホストのソフトウェアスタックは信頼しない．TEE 内ランタイムとプロセスのバイナリは信頼するものとし，その完全性はリモートアテステーションによって確認できることを想定する．一つの CVM 内では複数の Confidential Process が動作しうるが，これらは互いに信頼するものとする．例えば，fork によって生成された同一アプリケーションのプロセス群がこれに当たる．互いに信頼できない Confidential Process どうしは，それぞれ別の CVM で実行することを想定する．Confidential Process は信頼できないホストにシステムコールを委譲するため，ホストは悪意のある結果を返すことができ，Iago 攻撃 [12] の余地が生じる．その部分的な対策については 4.3 節で議論する．サイドチャネル攻撃および denial of service 攻撃は対象外とする．

3.2 システムコールの委譲

Confidential Process が発行した一部のシステムコールは，TEE の外で動作する**ホストプロキシプロセス**（以下，プロキシ）が Confidential Process に代わって実行する．ホスト側では，CVM ごとに一つの仮想マシンモニタ（VMM）プロセスと，Confidential Process ごとに一つのプロキシが動作する（**図 3**）．プロキシは対応する Confidential Process のシステムコール要求を処理し，open したファイルなど，そのプロセスに付随する OS 上の状態を保持する．VMM は すべての vCPU を実行するが，自身はシステムコール要求を処理せず，vCPU と対応するプロキシ間の中継のみを担う．VMM とプロキシはいずれも TEE の外にあり，信頼されない．

システムコールは次の流れで処理される．Confidential Process がシステムコールを発行すると，TEE 内ランタイムがこれを捕捉し，システムコールの番号と引数を共有メモリに書き込んでからホストに制御を移す．VMM は，そのvCPU に対応するプロキシに要求の到着を通知する．プロキシは共有メモリから要求を読み取ってシステムコールを実行し，結果を共有メモリに書き戻したうえで VMM に完了を通知する．VMM が vCPU の実行を再開すると，TEE 内ランタイムは結果を検証してから Confidential Process に結果を返す．`fork` や `exit` のようにプロセス（スレッドを含む）の生成や終了を伴うシステムコールでは，プロキシと VMM が協調して追加で必要な処理を行う．これにより，プロセスの生成や終了を経ても，Confidential Process とプロキシの一対一の対応関係を保つようにする．

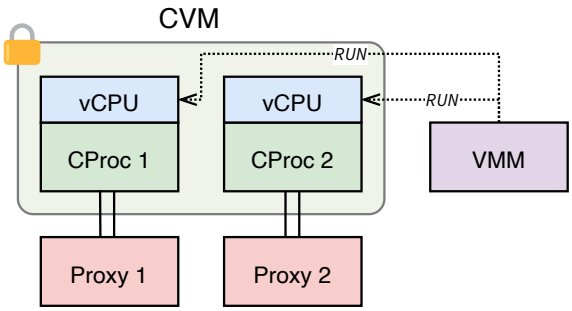

図 3 Confidential Process（CProc）とプロキシの対応関係．各 CProc は TEE 外の対応するプロキシにシステムコールを委譲する．VMM は CVM ごとに一つ存在し，vCPU の実行を担う．

3.3 TEE 内ランタイム

TEE 内には，Confidential Process とともに最小限のランタイムが置かれる．TEE 内ランタイムは，前述のとおりゲストの特権モードで動作し，ページテーブルや例外を管理するとともに，Confidential Process が発行したシステムコールを捕捉する．捕捉したシステムコールは，その種類に応じて TEE 内ランタイム自身が処理するか，プロキシに委譲する．プロキシから返された結果に対しては，そのシステムコールの意味論に基づいて基本的な検証を行う．これにより buffer overflow などを狙った Iago 攻撃を防ぐ．検証に成功した場合にのみ，Confidential Process の実行を再開し，もし検証に失敗したらプロセスを強制終了する．TEE 内ランタイムの役割は，CPU を含むハードウェアの制御，一部のシステムコールの処理，プロキシが返すシステムコールの結果の検証の三つに集約される．

3.4 システムコールの分類

本節では，どのようなシステムコールを TEE 内で直接処理する必要があり，どのようなシステムコールをホストに委譲できるかを明らかにする．ホストに委譲できない理由は，そのシステムコールが，ホストからは参照も変更もできない TEE 内の状態を扱うことにある．2 章で述べたとおり，ハイパーバイザは CVM のプライベートメモリやレジスタ状態にアクセスできないため，こうした状態を直接扱う必要がある処理はホストでは実行できない．

ⓒ2026 Information Processing Society of Japan

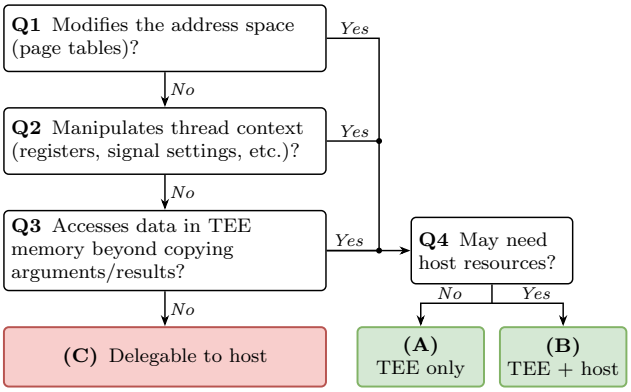

図 4 システムコールの処理場所の判定チャート．
Q1–Q3 のいずれかに該当すれば TEE 内ランタイムが処理し（A，B），いずれにも該当しなければホストに委譲できる（C）．判定はシステムコールと対象オブジェクトの組ごとに行う．

図 4 に，この基準を具体的にした判定チャートを示す．Q1 から Q3 は，TEE 内にしか存在しない三種類の状態，すなわちアドレス空間，スレッドのコンテキスト，TEE 内のメモリ上のデータ（引数と結果のコピーを除く）にそれぞれ対応する．いずれかに該当するシステムコールは TEE 内ランタイムが処理し，さらに Q4 によって，処理が TEE 内で完結するか（A），ホストを補助的に利用するか（B）に分かれる．Q1 から Q3 のいずれにも該当しないシステムコールは，ホストに委譲できる（C）．以下では，各経路の代表的なシステムコールについて述べる．

3.4.1 A: TEE 内で完結

匿名メモリに対する `mmap` は Q1 で分岐する．ページテーブルはプライベートメモリ上にあり，ホストからは書き換えられない．仮にこれを委譲してホストの指示通りにページテーブルを更新すれば，ホストは任意のページをプロセスのアドレス空間に挿入でき，CVM である意味がなくなる．

`rt_sigreturn` によるレジスタ状態の復元は Q2 で分岐する．ホストはレジスタ状態にアクセスできないため TEE 内で処理する．

`futex` は Q3 で分岐する．`futex` による待機では，futex word の値が期待値と一致することの確認と，スレッドの休止とを不可分に行う必要がある．futex word の値を共有メモリにコピーしてホストに委譲すると，lost wake-up の問題が生じる．このため，TEE 内ランタイムが待機キューを管理し，値の確認と休止を TEE 内で不可分に行う．

3.4.2 B: TEE 内で処理し，ホストを補助的に利用

`fork` は Q1 と Q2 に該当し，TEE 内ランタイムが親プロセスのアドレス空間とレジスタ状態を複製して子プロセスを生成する．一方，Confidential Process とプロキシの一対一の対応関係（3.2 節）を保つため，ホスト側でも子プロセスに対応するプロキシを用意する．

通常ファイルに対する `read` は Q3 で分岐する．ファイルを対象とする `mmap` では，そのファイルのページキャッシュをプライベートメモリ上に作成してマップするため，`read` がホストから直接ファイルを読むと，マップされた内容と一貫しなくなる．そのため，TEE 内ランタイムは通常ファイルに対する `read` をそのファイルのページキャッシュに対して処理する．ホストは，キャッシュにないページを読み込む際のバッキングストアとして利用される．なお，ソケットやパイプに対する `read` はページキャッシュを経由しないため C に該当し，経路は対象によっても異なることに注意されたい．

3.4.3 C: ホストに委譲可能

ソケット操作やパイプの操作，および `open` などのファイルシステム操作はホストに委譲可能である．ただし，送受信されるデータはホストを経由するため，TEE 内ランタイムはファイルディスクリプタの種類に応じて保護の方法を決める．パイプや UNIX ドメインソケットでは TEE 内ランタイムがデータを透過的に暗号化し，ネットワークソケットでは一般的な CVM と同様にアプリケーションが TLS などで保護する．

4. 実装

我々は Confidential Process を AMD SEV-SNP 上に Rust で実装した．プロトタイプは，TEE 内ランタイム，KVM 上に構築した独自の VMM，およびプロキシから構成される．現在のプロトタイプは 83 個のシステムコールに対応している．コードベース全体の大きさは約 18 KLOC であり，TCB に含まれる部分は 14 KLOC である．この TCB のコード量は，一般的な CVM で用いられる Linux カーネルより 2 桁小さく，Gramine-TDX [9] の 57 KLOC と比べても小さい．以下では，共有メモリの扱いとプロセスの起動手順について述べ，最後に互換性の範囲と今後の課題を整理する．

4.1 共有メモリの扱い

TEE 内ランタイムとプロキシは，共有メモリを介してシステムコールの要求と結果を受け渡す．システムコールの要求と結果の内容は，**システムコール記述子**と**システムコールバッファ**を用いて伝えられる．

システムコール記述子は，システムコールの種類，ファイルディスクリプタやサイズのような値そのものが意味を持つ引数，および戻り値を保持する．記述子の形式はシステムコールごとに定義し，TEE 内ランタイムとプロキシの間で共有する．各 vCPU が同時に実行するシステムコールは一つであるため，記述子は vCPU ごとに一つだけ共有メモリ上の決まった位置に置き，その vCPU で実行するすべてのシステムコールで使い回す．

一方，通常ポインタを介して受け渡される可変長のデータ（パス名や大きな構造体など）には，システムコール

ⓒ2026 Information Processing Society of Japan

バッファを用いる．Confidential Process が渡すポインタはプライベートメモリを指しており，当然ホストはその先を読み書きできないし，逆も然りである．そこで TEE 内ランタイムは，システムコールのたびに必要な大きさのシステムコールバッファを共有メモリ上の特定の領域から確保し，記述子にはポインタの代わりにこのバッファの位置を書き込む．システムコールへの入力となるデータがあれば，プロセスのメモリからバッファにコピーし，逆にシステムコールの結果のデータがあれば，バッファからプロセスのメモリにコピーする．プロキシはこのバッファをそのままホストのシステムコールの引数として使う．

例えば `open(pathname, flags, mode)` では，TEE 内ランタイムはまずパス名を格納できる大きさのシステムコールバッファを確保し，`pathname` の指すパス名をそこにコピーする．次に，その vCPU の記述子に `open` であること，`flags`，`mode`，およびバッファの位置を書き込み，ホストに制御を移す．プロキシはこのバッファを `pathname` としてホストの `open` を呼び出し，戻り値を記述子に書き込む．

なお，ホストは共有メモリの内容をいつでも書き換えられる．そのため TEE 内ランタイムは，記述子を共有メモリからプライベートメモリに一度だけコピーし，以降の検証と処理にはそのコピーのみを用いる．これにより，検証を終えた後に記述子を書き換えられても影響を受けず，TOCTOU 攻撃を防ぐ．

4.2 プロセスの起動

Confidential Process を起動する際，VMM はまず KVM を通じて SEV-SNP の CVM を作成し，TEE 内ランタイムのイメージとその実行に必要なページテーブルをプライベートメモリに配置する．VMM はこの領域のみを SEV-SNP のファームウェアに暗号化・計測させ，CVM を起動する．こうして得られる計測値が，アテステーションにおける信頼の基点となる．

起動時の計測対象を TEE 内ランタイムに限定する理由は二つある．一つ目は，計測値を実行するプログラムに依存させないためである．TEE 内ランタイムはどのプログラムを実行する場合も同一であるため，その測定値は固定であり，検証者は公開された単一の期待値と照合するだけで TEE 内ランタイムを検証できる．これは，固定された小さな起点から順に次の段階を計測していく計測ブート（measured boot）と同じ考え方である．二つ目は，起動時間を短く保つためである．SEV-SNP のファームウェアによる暗号化と計測は低速であり，その対象が大きいほど CVM の起動に時間がかかる [19]．対象を TEE 内ランタイムに限ることで，この処理は短時間で完了する．

プロセスのバイナリと引数は，ファームウェアの処理を経ずに受け渡せる共有メモリに配置される．CVM が起動すると，TEE 内ランタイムはメモリ管理などの実行環境を初期化した後，共有メモリ上の ELF バイナリをプライベートメモリに読み込んでプロセスイメージを構築し，プログラムのエントリポイントに制御を移す．プログラムのバイナリは信頼できないホストから渡されるため，TEE 内ランタイムによるバイナリの計測をリモートアテステーションに結び付ける仕組みが必要となるが，これは未実装である．

4.3 Confidential Process 特有の Iago 攻撃

Confidential Process はシステムコールを単位とするインターフェースでホストと通信するため，外部との入出力は基本的にファイルディスクリプタを介して行われる．データを保護する責任はファイルディスクリプタの種類によって異なり（3.4 節），データが正しく保護されるには，各ファイルディスクリプタが何を指すかについて，アプリケーションと TEE 内ランタイムの認識が一致していなければならない．

我々は Confidential Process に特有の Iago 攻撃として，ホストがスレッド同期を偽ってこの認識を食い違わせ，ネットワークソケットのような透過的暗号化の対象外のファイルディスクリプタを，UNIX ドメインソケットのような対象のファイルディスクリプタと取り違えさせる攻撃を特定した（付録 A.1）．この攻撃は，本質的にはファイルディスクリプタの再利用を突く．例えば，あるスレッドが UNIX ドメインソケットを閉じた後，別のスレッドがそれを知らずに同じファイルディスクリプタを使い続けてしまうとする．もしその番号が新たな TCP 接続に再利用されると，TEE 内ランタイムはその番号をネットワークソケットとして扱う一方，そのスレッドは UNIX ドメインソケットとして扱い続ける．その結果，そのスレッドが書き込んだ機密データは誰にも暗号化されないままホストに渡る．

そこで TEE 内ランタイムは，ファイルディスクリプタの番号の再利用を禁止する（**Stable File Descriptor Identity**）．閉じられた番号は使用済みとして占有し続け，新たなファイルディスクリプタには一度も使われていない番号を割り当てる．これにより，ファイルディスクリプタの再利用によって，同じ番号が実行中に別の種類のファイルディスクリプタを指すことを防ぐ．TEE 内ランタイムは，Confidential Process に見せるファイルディスクリプタをホスト側のものとは独立に割り当て，両者の対応を保持して，システムコールを委譲する際に相互に変換する．上記の規則を強制するのは前者に対してのみであり，ホスト側の番号は再利用されてもよい．暗号化の要否は TEE 内ランタイムが記録した種類によって決まり，ホスト側の番号には依存しないためである．

ⓒ2026 Information Processing Society of Japan

4.4 互換性の範囲と今後の課題

Confidential Process は，最小限の TEE 内ランタイムのもとで完全な POSIX 互換性を実現することを目指している．`execve` やパイプに対する透過的データ暗号化など，一部の機能はまだ実装されていないが，いずれも既存の機構の延長で実現できると考えている．これに対し，ファイルの暗号化とリモートアテステーションは本質的な課題として残っている．SCONE [7] ではパスごとに暗号化と認証の有無を選べるファイルシステムシールドを提供しており，Confidential Process でもホスト上に置かれたファイルに対して同等の保護が求められる．リモートアテステーションについては，TEE 内ランタイムのアテステーションを起点として，ユーザプログラムの完全性の確認やファイルの暗号鍵の配布をどのように実現するか，研究の余地がある．

5. 評価

本章では，Confidential Process の性能を次の 3 つの観点から評価する．

(1) システムコール委譲による実行時のオーバーヘッド
(2) 実アプリケーションの性能
(3) 起動から終了までのライフサイクルのコスト

実験には, AMD EPYC 9754 プロセッサ（SMT disabled, 2.25 GHz）と 128 GB のメモリを備えたマシンを用いた．ホスト OS は Ubuntu 24.04 LTS であり，カーネルには SNP ホスト向けの設定でビルドした Linux 6.17.7 を用いた．CVM のゲスト OS には Ubuntu 24.04 LTS（Linux 6.8.0）を用いた．すべての実験は同一のマシン上で行った．

5.1 マイクロベンチマーク

システムコールに焦点を当てたベンチマーク群である LMbench [13] を用いて，実行時のオーバーヘッドを評価した．比較対象は，同じベンチマークをホスト上で直接実行した結果である．**表 1** に実行結果を示す．各値は 10 回の計測の中央値である．右端の列は正規化した性能であり，レイテンシと帯域幅いずれの場合でもオーバーヘッドが小さいほど値が大きくなるようにした．

`lat_syscall null` は，ホストに委譲される `getppid` のレイテンシを計測するものであるが，Linux の約 85 倍の時間がかかった．このコストの大部分は `getppid` の処理そのものではなく，制御の遷移によるものである．VMEXIT/VMRUN によるゲストとホストの間の遷移に加え，eventfd を使った VMM とプロキシの間の遷移も存在するため，かなりのオーバーヘッドが生じている．一方，`lat_sig install` と `lat_pagefault` は CVM 内で大部分が完結するベンチマークであり，Linux を上回る性能を示した．帯域幅の正規化性能は 0.179 から 0.343 であり，`lat_syscall null` の 0.0118 より大幅に高い．これは，システムコール委譲の固定費が 64 KB のデータ転送に償却されるためである．ただし，64 KB のデータ転送でも，依然として固定費が支配的である．

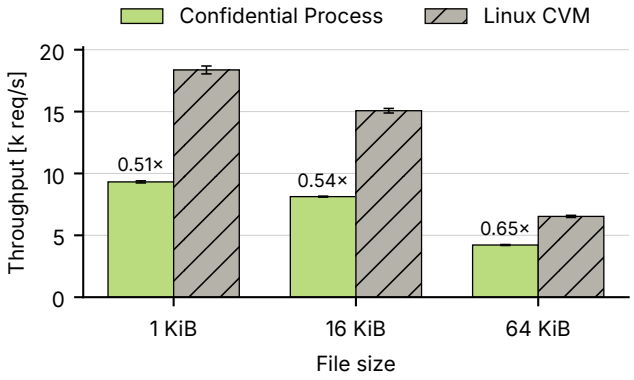

図 5 Nginx のスループット．Confidential Process の棒の上の数値は Linux CVM に対する相対スループットを示す．

システムコール委譲のオーバーヘッドを減らす策としては，KVM が提供する ioeventfd を用いて，VMM を介さずにホストカーネルからプロキシに直接通知する方法が考えられる．この場合，ゲストは基本的には HLT してプロキシによるシステムコール処理完了を待つことになるが，一定時間のスピン待機を許容すれば，HLT しない fastpath を設けることもできる．

5.2 マクロベンチマーク

実アプリケーションでの性能の評価として, Nginx（1.30.2）を用いてマクロベンチマークを行った．ApacheBench を使い同時接続数 16 で負荷を与え，リクエスト数のスループットを計測した．比較対象には，Linux CVM を用いる．CVM のネットワークには，オーバーヘッドの小さい vhost-net をバックエンドとする virtio-net で構成した．

Confidential Process のスループットは，Linux CVM の 51〜65% であった（**図 5**）．性能差の主な原因は，システムコール委譲のオーバーヘッドにあると考えられる．1 リクエストあたり平均して 2 回のシステムコール委譲（受信 `recvfrom` と送信 `writev`）が発生していると仮定すると，1 KiB でいえば，約半分の時間が委譲のオーバーヘッドである．ファイルサイズが大きいほど，相対的な差は縮まった．委譲のコストはリクエストあたりほぼ一定である一方，データの転送時間はファイルサイズに応じて増えるため，処理時間に占める委譲の割合が下がっている．

5.3 ライフサイクルコスト

ライフサイクルのコストを定量化するため，`main` 関数から 0 を返すだけの最小の Confidential Process について，起動から終了までのコストを計測した．計測区間は，起動側のプロセスが `fork` を呼び出してから `waitpid` が戻るまでであり，5 回の予備実行の後に 100 回計測した．比較

表 1 LMbench によるレイテンシと帯域幅の計測結果

	Command	Unit	Conf-proc	Native-Linux	Normalized[1]
Latency	`lat_syscall null`	µs	27.8	0.327	0.0118
	`lat_sig install`	µs	0.322	0.484	1.50
	`lat_pagefault`	µs	0.306	0.356	1.16
	`lat_mmap 4m`	µs	68.0	50.0	0.735
	`lat_proc fork`	ms	10.8	0.185	0.0171
	`lat_pipe`	µs	69.6	10.2	0.147
	`lat_tcp localhost`	µs	72.2[2]	23.4	0.324
Bandwidth[3]	`bw_file_rd 4m io_only`	GB/s	1.35	7.54	0.179
	`bw_pipe`	GB/s	0.735	2.38	0.309
	`bw_tcp localhost`	GB/s	0.724[2]	2.11	0.343

[1] Normalized performance. Native-Linux/Conf-proc for latency and Conf-proc/Native-Linux for bandwidth. Higher values indicate lower overhead.
[2] The confidential process communicates with an auxiliary process on the host side.
[3] The buffer in use is 64KB.

表 2 Confidential Process のライフサイクルのレイテンシ

		Mean	P99
Conf-proc	End-to-End (ms)	574	612
	CVM Creation (ms)	393	404
	Launch Measurement[1] (ms)	385	394
Linux CVM	End-to-End (ms)	3893	3936
	Launch Measurement[2] (ms)	1124	1150

[1] The total latency of the `KVM_SEV_SNP_LAUNCH_UPDATE` command [20].
[2] Estimated by scaling the Conf-proc launch measurement by the ratio of measured memory (4,284 KiB / 1,468 KiB).

対象の Linux CVM では，再起動を要求するだけのプログラムを init として実行し，同様にライフサイクルのコストを計測した．**表 2** に結果を示す．

Confidential Process の起動から終了までの時間は平均値で 574 ms であり，Linux CVM と比べて約 6.8 倍短い．内訳を見ると，CVM の作成に 393 ms を要し，そのうち 98% を AMD-SP による暗号化と計測が占める．Confidential Process の信頼の基点の大きさは 1,468 KiB であり，これには TEE 内ランタイムとその実行に必要な領域が含まれる．一方，TEE 内ランタイム単体の大きさは 428 KiB にとどまる．残りの 1 MiB 程度は，システムの初期化手順を最適化することで削減する余地があり，起動時間をさらに縮小できると考えている．

6. 関連研究

6.1 SGX における互換性

Intel SGX の互換性の問題に対し，エンクレーブ内に互換レイヤ，あるいはライブラリ OS を取り込むシステムが提案されている．

SCONE [7] は，SGX を用いて Docker コンテナ内のプロセスを保護するセキュアコンテナ機構である．エンクレーブ内に改変した libc を静的リンクし，システムコールはエンクレーブ外で実行する．この際，ファイルやネットワークの入出力は透過的に暗号化される．ユーザレベルスレッドのモデルと SGX のアーキテクチャ上の制約から，`fork` や `exec` はサポートしない．

Occlum [6] は，単一のエンクレーブ内で複数プロセスを安全かつ効率的に実行するためのライブラリ OS である．各プロセスを Software Fault Isolation（SFI）で隔離されたプロセスとして，ライブラリ OS と同一のアドレス空間に配置する．エンクレーブごとにプロセスを分離する従来方法と比べて，プロセス生成とプロセス間通信を高速化している．しかし，専用のバイナリ検証器によって確認済みの，静的リンクされたプログラムしか実行できない．`fork` もサポートしないため，`fork` を用いるアプリケーションは `spawn` を使うように書き換える必要がある．

これらのシステムはいずれもソースコードレベルの互換性にとどまる．また，SGX のアーキテクチャ上の制約により，プロセス管理に関する互換性の問題が依然として残っている．これに対し，Confidential Process は CVM を隔離基盤とするため，TEE 内で特権モードを利用できる．その結果，ページテーブルの管理など，プロセス管理に必要な処理を TEE 内で行えるため，SGX ベースのシステムと比べて ABI レベルでより高い互換性を確保できる．

6.2 CVM における TCB の削減

CVM では，ゲスト OS 全体が TCB に含まれる．この問題に対し，ゲスト内のカーネルを小さくする，あるいは CVM 内で特権分離を行うことで，TCB を削減するシステムが提案されている．

Gramine-TDX [9] は，Intel TDX 上で未改変の Linux

アプリケーションを実行するためのライブラリ OS である．ネットワークスタックとファイルシステムスタックをゲストから取り除き，準仮想化デバイスドライバを通じてそれらの処理をホストに委ねることで，TCB を約 57,000 行に抑えている．しかし，現状では単一プロセスのアプリケーションしか実行できない．CVM を隔離基盤として使った上で，I/O の処理をホストに委ねることで TCB を削減するという点で Confidential Process に近いアプローチである．両者の違いは，信頼境界を従来の VM インターフェースに置くか，システムコールインターフェースに置くかという点にある．

CVM 内の特権分離によって，ゲスト OS を TCB から除外する手法もある．Veil [10] は，AMD SEV-SNP の VMPL を用いて，CVM 内にゲスト OS より高い特権をもつセキュリティモニタを配置し，その上でゲスト OS からプログラムを保護するエンクレーブを提供する．しかし，このエンクレーブは SGX と同様のモデルに従うため，プログラムを専用の SDK で静的リンクし，ソースコードを修正する必要がある．また，シングルスレッドのプログラムしか実行できない．Wallet [11] は，サーバーレス環境向けに同様の特権分離を用いるシステムであり，各関数を trustlet と呼ぶ最小限のプロセスとして，ゲスト OS から隔離して実行する．これにより，CVM 上に関数をそのまま配置する場合と比べて，TCB を約 23% 削減している．しかし，trustlet はライブラリ OS の上で関数を実行するモデルで，実行環境に制限がある．Veil と Wallet はいずれも，TCB の削減と引き換えにアプリケーション互換性に制約を課している．

7. おわりに

本論文では，アプリケーション互換性と TCB 削減を両立する新たな TEE の抽象化として，Confidential Process を提案した．Confidential Process はプロセスを基本的な保護対象とし，CVM を隔離基盤として利用する．TEE 内には信頼されたプロセスと最小限のランタイムのみを配置し，システムコールに基づくインタフェースを通じてホスト OS とやり取りする．これにより，ゲスト OS 全体を TCB に含めることなく，既存アプリケーションの無改変実行を可能にする．AMD SEV-SNP 上に実装したプロトタイプの TCB は約 14 KLOC であり，起動から終了までの時間は Linux CVM と比べて約 6.8 倍短かった．一方，実行時にはシステムコール委譲に伴う制御の遷移が主なボトルネックとなり，Nginx のスループットは Linux CVM の 51〜65%にとどまった．

今後の課題は次のとおりである．性能面では，ioeventfd を用いて VMM を介さずにプロキシへ通知する経路やスピン待機による fastpath を導入し，システムコール委譲のコストを削減する．機能面では，`execve` やパイプに対する透過的暗号化を実装し，POSIX 互換性を高める．さらに，ホスト上のファイルの保護と，リモートアテステーションによってユーザープログラムの完全性を確認する仕組みについて検討を進める．

参考文献

[1] Schlüter, B., Sridhara, S., Kuhne, M., Bertschi, A. and Shinde, S.: HECKLER: Breaking Confidential VMs with Malicious Interrupts, *Proceedings of the 33rd USENIX Security Symposium (USENIX Security 24)*, pp. 3459–3476 (2024).

[2] Schlüter, B., Sridhara, S., Bertschi, A. and Shinde, S.: WeSee: Using Malicious #VC Interrupts to Break AMD SEV-SNP, *Proceedings of the 2024 IEEE Symposium on Security and Privacy (SP)*, pp. 4220–4238 (online), DOI: 10.1109/SP54263.2024.00262 (2024).

[3] Takekoshi, S., Mori, M., Fukai, T. and Shinagawa, T.: BadAML: Exploiting Legacy Firmware Interfaces to Compromise Confidential Virtual Machines, *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, CCS '25, p. 4469–4483 (online), DOI: 10.1145/3719027.3765123 (2025).

[4] Schlüter, B. and Shinde, S.: RMPocalypse: How a Catch-22 Breaks AMD SEV-SNP, *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, CCS '25, p. 3840–3854 (online), DOI: 10.1145/3719027.3765233 (2025).

[5] Baumann, A., Peinado, M. and Hunt, G.: Shielding Applications from an Untrusted Cloud with Haven, *ACM Trans. Comput. Syst.*, Vol. 33, No. 3 (online), DOI: 10.1145/2799647 (2015).

[6] Shen, Y., Tian, H., Chen, Y., Chen, K., Wang, R., Xu, Y., Xia, Y. and Yan, S.: Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX, *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, p. 955–970 (online), DOI: 10.1145/3373376.3378469 (2020).

[7] Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M. L., Goltzsche, D., Eyers, D., Kapitza, R., Pietzuch, P. and Fetzer, C.: SCONE: Secure Linux Containers with Intel SGX, *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 689–703 (online), available from ⟨https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov⟩ (2016).

[8] Shinde, S., Le, D., Tople, S. and Saxena, P.: Panoply: Low-TCB Linux Applications with SGX Enclaves, *Proceedings of the 24th Annual Network and Distributed System Security Symposium*, (online), DOI: 10.14722/ndss.2017.23500 (2017).

[9] Kuvaiskii, D., Stavrakakis, D., Qin, K., Xing, C., Bhatotia, P. and Vij, M.: Gramine-TDX: A Lightweight OS Kernel for Confidential VMs, *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, p. 4598–4612 (online), DOI: 10.1145/3658644.3690323 (2024).

[10] Ahmad, A., Ou, B., Liu, C., Zhang, X. and Fonseca, P.: Veil: A Protected Services Framework for Confidential Virtual Machines, *Proceedings of the 28th ACM International Conference on Architectural Sup-*

ⓒ2026 Information Processing Society of Japan

port for Programming Languages and Operating Systems, Volume 4, ASPLOS '23, p. 378–393 (online), DOI: 10.1145/3623278.3624763 (2024).

[11] Sabanic, P., Misono, M., Bodea, T., Pritzi, J., Hackl, M., Stavrakakis, D. and Bhatotia, P.: Wallet: Confidential Serverless Computing, *Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation* (2026).

[12] Checkoway, S. and Shacham, H.: Iago attacks: why the system call API is a bad untrusted RPC interface, *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, p. 253–264 (online), DOI: 10.1145/2451116.2451145 (2013).

[13] McVoy, L. and Staelin, C.: lmbench: Portable Tools for Performance Analysis, *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, USA, USENIX Association, p. 23 (1996).

[14] Costan, V. and Devadas, S.: Intel SGX Explained, *IACR Cryptology ePrint Archive* (2016).

[15] Advanced Micro Devices, Inc.: AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More, Technical report, Advanced Micro Devices, Inc. (2020).

[16] Intel Corporation: Intel Trust Domain Extensions (Intel TDX) White Paper (2025). Document Number 343961-003US.

[17] Li, X., Li, X., Dall, C., Gu, R., Nieh, J., Sait, Y. and Stockwell, G.: Design and Verification of the Arm Confidential Compute Architecture, *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, Carlsbad, CA, USENIX Association, pp. 465–484 (2022).

[18] Misono, M., Stavrakakis, D., Santos, N. and Bhatotia, P.: Confidential VMs Explained: An Empirical Analysis of AMD SEV-SNP and Intel TDX, *Proc. ACM Meas. Anal. Comput. Syst.*, Vol. 8, No. 3 (online), DOI: 10.1145/3700418 (2024).

[19] Holmes, B., Waterman, J. and Williams, D.: SEVeriFast: Minimizing the root of trust for fast startup of SEV microVMs, *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, p. 1045–1060 (online), DOI: 10.1145/3620665.3640424 (2024).

[20] The kernel development community: Secure Encrypted Virtualization (SEV) — The Linux Kernel documentation, https://docs.kernel.org/6.17/virt/kvm/x86/amd-memory-encryption.html (2025).

付　　録

A.1 ファイルディスクリプタの再利用を突く攻撃の例

図 A·1 に，4.3 節で述べた攻撃の対象となるプログラムの例を示す．このプログラムは，TCP の待ち受けソケット `listen_fd` と UNIX ドメインソケット `uds_fd` を，`EPOLLONESHOT` を指定して epoll に登録し，複数のスレッドで `worker` を並行に実行する．`EPOLLONESHOT` を指定したファイルディスクリプタは，一度イベントを通知すると，`EPOLL_CTL_MOD` で再び有効化されるまで通知されない．そのため正直なカーネルのもとでは，`uds_fd` を処理するスレッドは常に一つであり，(3) で閉じた後に他のスレッドが `uds_fd` を使うことはない．

```
// epfd: epoll FD
// listen_fd: TCP listening socket
// uds_fd: UNIX domain socket

static void arm(int op, int fd) {
  struct epoll_event ev = {
    .events = EPOLLIN | EPOLLONESHOT,
    .data.fd = fd,
  };
  epoll_ctl(epfd, op, fd, &ev);
}

// Multiple threads execute this function concurrently.
// Before spawning them, the main thread registers
// listen_fd and uds_fd with epoll by calling
// arm(EPOLL_CTL_ADD, ...).
void worker(void) {
  struct epoll_event e;
  char buf[4096];
  for (;;) {
    epoll_wait(epfd, &e, 1, -1);
    int fd = e.data.fd;
    if (fd == listen_fd) {
      int cfd = accept(listen_fd, NULL, NULL);   // (1)
      arm(EPOLL_CTL_ADD, cfd);
      arm(EPOLL_CTL_MOD, listen_fd);
    } else if (fd == uds_fd) {
      if (read(uds_fd, buf, sizeof(buf)) == 0) { // (2)
        epoll_ctl(epfd, EPOLL_CTL_DEL, uds_fd, NULL);
        close(uds_fd);                           // (3)
        continue;
      }
      /* ... */
      write(uds_fd, secret, len);                // (4)
      arm(EPOLL_CTL_MOD, uds_fd);
    } else {
      /* Handle TCP clients. */
    }
  }
}
```

図 A·1 ファイルディスクリプタの再利用を突く攻撃の対象となるプログラム．

悪意あるホストは，`EPOLLONESHOT` を無視して，`uds_fd` の同じイベントを複数のスレッドの `epoll_wait` に返すことができる．スレッド A と B が同時に `uds_fd` を処理し，(2) で B はデータを受け取り，A は 0 を受け取ったとする．A が (3) で `uds_fd` を閉じた後，スレッド C が (1) で新たな TCP 接続を受け付けると，閉じられた番号がこの接続に再利用されうる．TEE 内ランタイムは，最新の情報に基づいてこの番号を TCP 接続とみなす．しかし B は，この番号を UNIX ドメインソケットとみなしたまま，(4) で機密データを書き込む．アプリケーションは TEE 内ランタイムが暗号化すると想定し，TEE 内ランタイムはア

プリケーションが保護すると想定するため，データは平文のままホストに渡る．

Stable File Descriptor Identity のもとでは，(3) で閉じた番号は使用済みとして占有され続け，(1) で受け付けた接続には別の番号が割り当てられる．そのため，(4) の `write` は使用済みの番号に対する操作としてエラーとなり，データは漏洩しない．

ⓒ2026 Information Processing Society of Japan